

# Docker 操作手册

# 目 录

1. docker 概述.....	5
1.1 docker 是什么.....	5
1.2 为什么使用 docker.....	5
2. docker 原理与架构.....	5
2.1 镜像的原理.....	5
2.2 镜像与容器的关系.....	6
3. 安装.....	7
3.1 前提条件.....	7
3.2 使用 yum 安装.....	7
3.2.2 设置 yum 源.....	7
3.3.3 查看所有仓库中的 docker 版本.....	7
3.3.4 安装指定的版本.....	8
3.3.5 启动并加入开机自启.....	8
3.3.6 验证安装是否成功.....	8
3.3.7 没有网络的情况下离线安装.....	8
3.4 配置阿里云镜像仓库.....	8
3.4.1 打开阿里云控制台.....	9
3.4.2 配置方法如下，参照阿里云的说明即可.....	9
3.5 设置镜像容器存放的默认路径.....	9
3.5.1 docker 启动之前可修改 docker.conf 中的镜像路径.....	10
3.5.2 镜像容器有数据时修改路径.....	10
3.6 swarm 集群搭建.....	10
3.6.1 前提条件.....	10
3.6.2 集群部署.....	10
3.7 集群命令.....	11
3.7.1 初始化一个节点并绑定网卡地址.....	11
3.7.2 查看加入集群并成为管理节点的命令.....	11
3.7.3 查看加入集群并成为工作节点的命令.....	11
3.7.4 使旧令牌无效并生成新令牌.....	11
3.7.5 查看集群中的节点.....	12

3.7.6 更新节点的状态 (active, pause, drain) .....	12
3.7.7 将节点升级为 manager .....	12
3.7.8 将节点降级为 worker .....	12
4. 制作镜像-DockerFile .....	12
4.1 配置镜像仓库 .....	12
4.2 搜索/拉取镜像 .....	12
4.3 制作镜像-编写 dockerfile .....	12
4.4 dockerfile 样例文件: .....	17
5. 镜像与容器命令 .....	18
5.1 镜像相关命令 .....	18
5.1.1 获取镜像 .....	18
5.1.2 列出本地镜像 .....	18
5.1.3 把容器保存为镜像 .....	18
5.1.4 删除本地镜像 .....	18
5.1.5 删除为 none 的镜像 .....	19
5.1.6 修改镜像名 .....	19
5.1.7 导出镜像为 .....	19
5.1.8 导入镜像 .....	19
5.1.9 搜索镜像仓库中的镜像 .....	19
5.1.10 构建镜像 (dockerfile 文件在本地目录) .....	19
5.1.11 显示镜像的历史更改记录 .....	19
5.2 容器相关命令 .....	19
5.2.1 进入容器 .....	19
5.2.2 复制文件到容器 .....	19
5.2.3 从容器复制文件到本地 .....	19
5.2.4 停止已退出的 docker .....	19
5.2.5 删除已退出的容器 .....	19
5.2.6 查看日志 .....	19
5.2.7 docker events .....	19
5.2.8 docker 监控命令 .....	20
5.2.9 查看镜像或容器的信息 .....	20
5.2.10 查看容器的端口映射 .....	20

5.2.11 显示所有运行和停止的 docker.....	20
5.2.12 通过 filter 查找退出状态值为 0 的所有 docker.....	20
5.2.13 重启停止的 docker.....	20
5.3 docker run 命令.....	20
5.3.1 docker 镜像 nginx:latest 后台启动 nginx.....	20
5.3.2 使用镜像 nginx:latest 以后台模式启动一个容器, 并将容器的 80 端口映射到主机随机端口。.....	21
5.3.3 使用镜像 nginx:latest, 以后台模式启动一个容器, 将容器的 80 端口映射到主机的 8000 端口, 主机的目录 /data 映射到容器的 /data。.....	21
5.3.4 绑定容器的 8080 端口, 并将其映射到本地主机 127.0.0.1 的 80 端口上。.....	21
5.3.5 使用镜像 nginx:latest 以交互模式启动一个容器, 在容器内执行/bin/bash 命令。.....	21
5.4 docker 系统管理命令.....	21
5.4.1 查看 docker 磁盘使用状况.....	21
5.4.2 清理磁盘, 删除关闭的容器、无用的数据卷和网络, 以及 dangling 镜像(即无 tag 的镜像)。.....	21
5.4.3 清理得更加彻底, 可以将没有容器使用 Docker 镜像都删掉。注意, 这两个命令会把你暂时关闭的容器, 以及暂时没有用到的 Docker 镜像都删掉了...所以使用之前一定要想清楚呐。.....	21
6. compose (以 V3 为例) .....	22
6.2 compose 参考文件.....	29
7. docker compose 命令.....	30
7.1 部署 docker 集群.....	30
7.2 列出所有节点.....	30
7.3 列出所有 stack.....	30
7.4 列出 stack 中所有任务.....	30
7.5 删除 stack.....	30
7.6 列出 stack 中所有服务.....	30
7.7 查看网卡信息.....	30
7.8 更新服务所在的节点.....	30
7.9 列出所有的服务.....	30
8. 问题汇总.....	31

8.1 如何批量清理临时镜像文件.....	31
8.2 运行 docker 容器，出现如下错误.....	31
8.3 docker 无法删除本地镜像.....	31
8.4 容器中的时区与语言的问题.....	32
8.5、docker 启动报错 error initializing graphdriver.....	32
8.6 iptables failed.....	33
8.7 Unable to take ownership of thin-pool.....	34
8.8 docker 启动报错：.....	34
8.9 问题 docker dead but pid file exists.....	34
8.10 使用/bin/bash 无法进入容器.....	35
8.11 compose 暴漏端口问题.....	35
8.12 docker 中部署的程序刚起来后，来不及看日志及调试 docker 就挂掉了.....	35
8.13 stack 集群模式启动后，修改 config 标签定义的配置文件，删除单个容器后，再重启该容器发现配置未生效。.....	36
8.14 dockerfile 中 cmd 命令指定了容器的启动命令，但是 docker run -it container /bin/sh 启动容器后发现 dockerfile 中指定的命令并未运行.....	36
8.15 redis 启动时的警告优化.....	36
8.16 在 docker 容器中使用 FTP 客户端的问题.....	36
9. 注意事项.....	39
9.1 不要在容器中存储数据.....	39
9.2 不要发布两份应用.....	39
9.3 不要创建超大镜像.....	39
9.4 不要使用单层镜像.....	40
9.5 不要使用运行中的容器创建镜像.....	40
9.6 不要只使用“最新”标签.....	40
9.7 不要在单一容器中运行超过一个进程.....	40
9.8 不要在镜像中存储凭据，要使用环境变量.....	40
9.9 使用非 root 用户运行进程.....	41
9.10 不要依赖 IP 地址.....	41

# 1. docker 概述

## 1.1 docker 是什么

docker 是一个轻量级的容器，可以理解为经过优化处理的、丢掉没用的 90%的虚拟机垃圾之后，剩下的一个小型的容器；与虚拟机的不同之处在于 docker 没有对硬件虚拟化，docker 的硬件资源可动态分配调整。

## 1.2 为什么使用 docker

提供与物理机完全隔离的、可移植的运行环境，可以打包应用及应用所需的依赖到容器中，然后发布到任意的 linux 服务器，而不需担心 linux 服务器的环境

快速的创建和部署，容器可以秒级启动，且在不需要的时候可以快速移除

有效利用资源，因为 docker 经过极大的精简处理，几乎不占用系统资源，而 docker 中应用所需的资源又可以根据物理机资源动态分配

测试环境更接近于生产环境，因 docker 的隔离性，可将多个 docker 及装载的一系列服务部署于单机来模拟生产环境

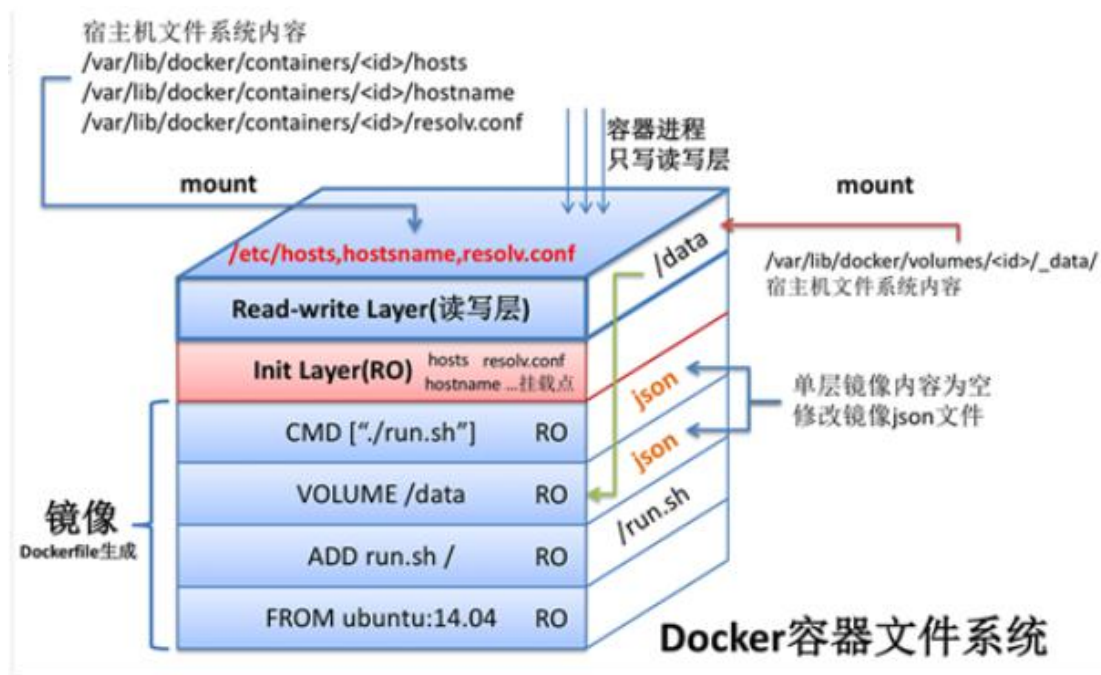
# 2. docker 原理与架构

## 2.1 镜像的原理

Dockerfile 是软件的原材料，Docker 镜像是软件的交付品，而 Docker 容器则可以认为是软件的运行态。从应用软件的角度来看，Dockerfile、Docker 镜像与 Docker 容器分别代表软件的三个不同阶段，Dockerfile 面向开发，Docker 镜像成为交付标准，Docker 容器则涉及部署与运维，三者缺一不可，合力充当 Docker 体系的基石。

Dockerfile 构建出 Docker 镜像，通过 Docker 镜像运行 Docker 容器。

我们可以从 Docker 容器的角度，来反推三者的关系。首先可以来看下图：



## 2.2 镜像与容器的关系

转化的依据是每个镜像的 json 文件，Docker 可以通过解析 Docker 镜像的 json 的文件，获知应该在这个镜像之上运行什么样的进程，应该为进程配置怎么样的环境变量，此时也就实现了静态向动态的转变。

谁来执行这个转化工作？答案是 Docker 守护进程。也许大家早就理解这样一句话：Docker 容器实质上就是一个或者多个进程，而容器的父进程就是 Docker 守护进程。这样的，转化工作的执行就不难理解了：Docker 守护进程手握 Docker 镜像的 json 文件，为容器配置相应的环境，并真正运行 Docker 镜像所指定的进程，完成 Docker 容器的真正创建。

Docker 容器运行起来之后，Docker 镜像 json 文件就失去作用了。此时 Docker 镜像的绝大部分作用就是：为 Docker 容器提供一个文件系统的视角，供容器内部的进程访问文件资源。

再次回到上图，我们再来看看容器和镜像之间的一些特殊关系。首先，之前已经提及 Docker 镜像是分层管理的，管理 Docker 容器的时候，Docker 镜像仍然是分层管理的。由于此时动态的容器中已经存在进程，进程就会对文件系统视角内的文件进行读写操作，因此，就会涉及一个问题：容器是否会篡改 Docker 镜像的内容？

答案自然是不会的。统一来讲，正如图，所有的 Docker 镜像层对于容器来说，都是只读的，容器对于文件的写操作绝对不会作用在镜像中。

## 3. 安装

### 3.1 前提条件

docker 是基于 Linux 64bit 的，无法在 32bit 的 linux/Windows/unix 环境下使用；

若在 linux6.X 上安装 docket，需升级内核版本至少 3.8 以上；

若使用 docker stack 模式部署，使用 linux7 系统；

建议使用 linux 版本 7 安装 docker。

### 3.2 使用 yum 安装

#### 3.2.2 设置 yum 源

yum-config-manager --add-repo <https://download.docker.com/linux/centos/docker-ce.repo>  
若没有 yum-config-manager 命令，则安装：

```
yum -y install yum-utils
```

#### 3.3.3 查看所有仓库中的 docker 版本

```
yum list docker-ce --showduplicates | sort -r
```

```
[root@master ~]# yum list docker-ce --showduplicates | sort -r
已加载插件: fastestmirror
已安装的软件包
可安装的软件包
 * updates: mirror.bit.edu.cn
Loading mirror speeds from cached hostfile
 * extras: mirror.bit.edu.cn
 * epel: mirrors.huaweicloud.com
docker-ce.x86_64            18.06.0.ce-3.el7           docker-ce-stable
docker-ce.x86_64            18.03.1.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            18.03.0.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.12.1.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.12.0.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.09.1.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.09.0.ce-1.el7.centos    installed
docker-ce.x86_64            17.09.0.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.06.2.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.06.1.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.06.0.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.03.2.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.03.1.ce-1.el7.centos    docker-ce-stable
docker-ce.x86_64            17.03.0.ce-1.el7.centos    docker-ce-stable
 * base: mirror.bit.edu.cn
[root@master ~]#
```



### 3.3.4 安装指定的版本

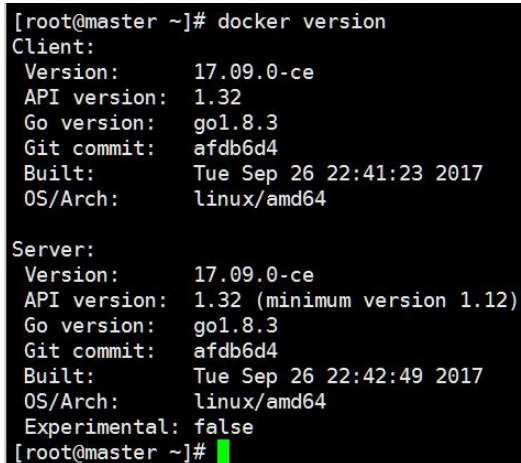
```
yum install docker-ce-18.06.1.ce
```

### 3.3.5 启动并加入开机自启

```
systemctl start docker  
systemctl enable docker
```

### 3.3.6 验证安装是否成功

```
docker version
```



```
[root@master ~]# docker version  
Client:  
Version:      17.09.0-ce  
API version:  1.32  
Go version:   go1.8.3  
Git commit:   afdb6d4  
Built:        Tue Sep 26 22:41:23 2017  
OS/Arch:      linux/amd64  
  
Server:  
Version:      17.09.0-ce  
API version:  1.32 (minimum version 1.12)  
Go version:   go1.8.3  
Git commit:   afdb6d4  
Built:        Tue Sep 26 22:42:49 2017  
OS/Arch:      linux/amd64  
Experimental: false  
[root@master ~]#
```

### 3.3.7 没有网络的情况下离线安装

(1) 先到官方地址下载所需的安装包，并上传到服务器

[https://download.docker.com/linux/centos/7/x86\\_64/stable/Packages/](https://download.docker.com/linux/centos/7/x86_64/stable/Packages/)

(2) `rpm -ivh docker-ce-*.rpm`

### 3.3.8 配置用户

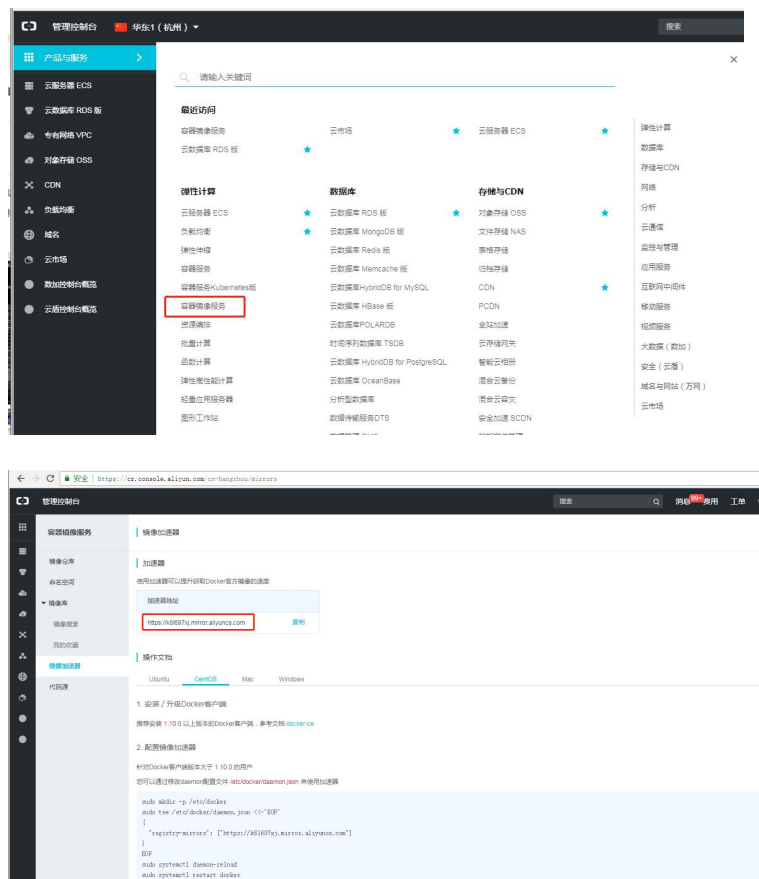
```
sudo groupadd docker #添加 docker 用户组 (默认会自动建)  
sudo gpasswd -a $USER docker #将登陆用户加入到 docker 用户组中  
newgrp docker #更新用户组  
docker ps #测试 docker 命令是否可以使用 sudo 正常使用
```

## 3.4 配置阿里云镜像仓库

因官方仓库下载速度慢，需配置国内镜像地址

### 3.4.1 打开阿里云控制台

没有的可以用淘宝账号或者支付宝账号直接登录



### 3.4.2 配置方法如下，参照阿里云的说明即可

```
sudo mkdir -p /etc/docker
```

```
sudo tee /etc/docker/daemon.json <<-'EOF' { "registry-mirrors":  
["https://k81697xj.mirror.aliyuncs.com"] } EOF sudo systemctl  
daemon-reload sudo systemctl restart docker
```

## 3.5 设置镜像容器存放的默认路径

### 3.5.1 docker 启动之前可修改 docker.conf 中的镜像路径

vi /etc/systemd/system/docker.service.d/docker.conf

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd --debug=false -s overlay --graph=/opt1/docker/images --insecure-registry docker.16qian.cn:5000 -H unix:///var/run/docker.sock -H tcp://10.95.5.251:2371
```

### 3.5.2 镜像容器有数据时修改路径

①docker info 查看默认保存路径: Docker Root Dir

例如:

原路径: /data/docker/images

目的路径: /opt1/docker/images

②mv /data/docker/images /opt1/docker/

③修改--graph 参数的路径为目标路径

vi /etc/systemd/system/docker.service.d/docker.conf

--graph=/opt1/docker/images

④重启 docker

systemctl daemon-reload

service docker restart

## 3.6 swarm 集群搭建

### 3.6.1 前提条件

docker 版本需要 1.12+及以上;

需要在搭建集群的各宿主机装好 docker

### 3.6.2 集群部署

① 在主节点上执行以下命令, 并保存结果信息

docker swarm init --advertise-addr 172.16.60.95

```
[root@master ~]# docker swarm init --advertise-addr 172.16.60.95
Swarm initialized: current node (kfi2r4dw6895z5yvh1byzfck6) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-3fzyz5knfbhw9iq1zxhb6dmzdr0izno9nr7iqc5wid09ug1h8-0mocmawzvm3xge6s37n5a48fw 172.16.60.95:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

## ② 使用 docker info 或 docker node ls 查看集群信息

```
[root@master ~]# docker info
Containers: 9
  Running: 2
  Paused: 0
  Stopped: 7
Images: 23
Server Version: 17.09.0-ce
Storage Driver: overlay
  Backing Filesystem: extfs
  Supports d_type: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: bridge host macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: active
NodeID: lylghiz2aeub0mecrphxv71nw
Is Manager: true
ClusterID: l467f1uh2uhjycoqbz4xqldkk
Managers: 1
Nodes: 1
Orchestration:
  Task History Retention Limit: 5
Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
```

## ③ 添加节点到 swarm 集群中(命令来自 swarm 初始化的结果)

```
docker swarm join --token
SWMTKN-1-3fzyz5knfbhw9iqzlzxb6dmzdtr0izno9nr7iqc5wid09uglh8-0mocmawzvm3xge6s37n5a4
8fw 172.16.60.95:2377
```

### 3.6.3 获取集群 token

swarm join-token : 可以查看或更换 join token。

docker swarm join-token worker: 查看加入 woker 的命令。

docker swarm join-token manager: 查看加入 manager 的命令

docker swarm join-token --rotate worker: 重置 woker 的 Token。

docker swarm join-token -q worker: 仅打印 Token。

## 3.7 集群命令

### 3.7.1 初始化一个节点并绑定网卡地址

```
docker swarm init --advertise-addr 192.168.10.111
```

### 3.7.2 查看加入集群并成为管理节点的命令

```
docker swarm join-token manager
```

### 3.7.3 查看加入集群并成为工作节点的命令

```
docker swarm join-token worker
```

### 3.7.4 使旧令牌无效并生成新令牌

```
docker swarm join-token --rotate
```

### 3.7.5 查看集群中的节点

```
docker node ls
```

### 3.7.6 更新节点的状态 (active, pause, drain)

```
docker node update --availability active docker-118
```

--active 此节点成为工作节点，并接受新任务

--pause 暂停为此节点分配新任务但已有的任务保持运行

--drain 不会为此节点分配新任务，并迁移现有的任务到其它节点

### 3.7.7 将节点升级为 manager

```
docker node promote docker-118
```

### 3.7.8 将节点降级为 worker

```
docker node demote docker-118
```

## 4. 制作镜像-DockerFile

### 4.1 配置镜像仓库

### 4.2 搜索/拉取镜像

### 4.3 制作镜像-编写 dockerfile

#### FROM

指定引用的基础镜像

推荐使用 [Alpine](#) 镜像，因为它尺寸较小（目前小于 5 MB），同时仍然是完整的 Linux 发行版。

#### LABEL

添加项目中 docker 镜像的版本信息，使用 `docker inspect` 可查看镜像的相关信息

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

例如：

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

#### EXPOSE

该 EXPOSE 指令通知 docker 容器在运行时侦听指定的网络端口，并不是向外部暴露端口。

要在 TCP 和 UDP 上公开，需要包含两行：

```
EXPOSE 80/tcp
```

EXPOSE 80/udp

## ENV

格式:

```
ENV <key> <value>
```

```
ENV <key>=<value> ..
```

说明:

说明: 定义环境变量的同时, 可以引用已经定义的环境变量。

在 ENV 指令中, 可以直接引用如下环境变量:

HOME, 用户主目录

HOSTNAME, 默认容器的主机名

PATH, 环境变量

TERM, 默认 xterm

示例

```
ENV PATH /usr/local/bin:$PATH
```

```
ENV LANG C.UTF-8
```

```
ENV TERM xterm
```

```
ENV PYTHON_VERSION 3.5.3
```

```
ENV name1=ping name2=on_ip
```

## ADD/COPY

格式:

```
ADD/COPY <src>... <dest>
```

```
ADD/COPY ["<src>","... "<dest>"]
```

说明与示例:

1、<src>路径必需在构建的上下文中; 不能使用 `ADD ../something /something`, 这是因为 `docker build` 的第一步就是发送上下文目录给 `docker daemon`。

2、每个都<src>可能包含通配符, 匹配将使用 Go 的 `filepath.Match` 规则完成。例如

```
ADD hom* /mydir/      # 以 hom 开头的所有文件
```

```
ADD hom?.txt /mydir/  # ?代替一个字符, eg: "home.txt"
```

3、<dest>是一个绝对路径, 或相对于一个路径 `WORKDIR`, 到其中的源将在目标容器内进

行复制。（工作目录使用 `docker inspect` 查看）

```
ADD test relativeDir/          # 添加"test"到工作目录/relativeDir/
ADD test /absoluteDir/        # 添加"test"到绝对路径/absoluteDir/
```

## CMD

格式:

```
CMD ["executable","param1","param2"]  #这是首选形式
CMD ["param1","param2"]               #作为 ENTRYPOINT 命令的参数
CMD command param1 param2             #shell 形式
```

说明:

CMD 为容器启动时提供默认的执行命令;

dockerfile 中只需定义一个 CMD，如果定义多个，只有最后一个生效;

如果 CMD 用于为 ENTRYPOINT 指令提供默认参数，则应使用 JSON 数组格式指定 CMD 和 ENTRYPOINT 指令;

与 *shell* 表单不同，*exec* 表单不会调用命令 *shell*。这意味着不会发生正常的 *shell* 处理。例如，CMD [ "echo", "\$HOME" ] 不会对变量进行替换 \$HOME。如果你想要 *shell* 处理，那么要么使用 *shell* 表单，要么直接执行 *shell*

## ENTRYPOINT

格式:

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2
```

说明:

设置容器启动时默认执行的命令，如果有多个 ENTRYPOINT，则最后一条生效，docker run 之后执行的命令作为 ENTRYPOINT 内容的参数

示例:

Dockerfile 设定: ENTRYPOINT ["/bin/echo"]

那么 docker build 出来的镜像以后的容器功能就像一个 /bin/echo 程序:

比如我 build 出来的镜像名称叫 imageecho，那么我可以这样用它:

docker run -it imageecho "this is a test"

这里就会输出” this is a test” 这串字符，而这个 imageecho 镜像对应的容器表现出来的功能就像一个 echo 程序一样。你添加的参数 “this is a test” 会添加到 ENTRYPOINT 后面，就成了这样 /bin/echo “this is a test” 。

## RUN

格式：

```
RUN <command>
```

```
RUN ["executable", "param1", "param2"]
```

说明：

在 *shell* 形式中，您可以使用\（反斜杠）将单个 RUN 指令继续到下一行。例如：

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
```

## VOLUME

格式：

```
VOLUME ["/data"]
```

说明：

为了能够保存（持久化）数据以及共享容器间的数据，Docker 提出了 Volume 的概念。简单来说，Volume 就是目录或者文件，它可以绕过默认的联合文件系统，而以正常的文件或者目录的形式存在于宿主机上。

在 dockerfile 中更改已经声明的 volume 是无效的；dockerfile 无法指定宿主机的目录，必须在创建或运行时指定宿主机的目录

样例：

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

## USER

格式：

```
USER <user>[:<group>] or
```

```
USER <UID>[:<GID>]
```

作用：

指定运行时的用户名或 UID，后续的 RUN 也会使用指定的用户。



当服务不需要管理权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户

说明：

要临时获取管理权限可以使用 `gosu`，而不推荐 `sudo`。

示例：

```
FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

## WORKDIR

格式：

```
WORKDIR /path/to/workdir
```

说明：

为后续的 `RUN`、`CMD`、`ENTRYPOINT` 指令配置工作目录。

可以使用多个 `WORKDIR` 指令，后续命令如果参数时相对路径，则会基于之前命令指定的路径

例如：

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
则最终路径为/a/b/c。
可以解析之前设置的环境变量，例如：
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
Pwd 命令的输出结果是/path/$DIRNAME
```

## ARG

格式：

```
ARG <name>[=<default value>]
```

说明：

lag. ARG 指令定义了用户可以在编译时或者运行时传递的变量，如使用如下命令：

```
--build-arg <varname>=<value>
```

示例：

```
FROM busybox
```

```
ARG user1
```

```
ARG buildno
```

Onbuild

格式：

```
ONBUILD [INSTRUCTION]
```

说明：

ONBUILD 指令可以为镜像添加触发器。其参数是任意一个 Dockerfile 指令。

当我们在一个 Dockerfile 文件中加上 ONBUILD 指令，该指令对利用该 Dockerfile 构建镜像（比如为 A 镜像）不会产生实质性影响。但是当我们编写一个新的 Dockerfile 文件来基于 A 镜像构建一个镜像（比如为 B 镜像）时，这时构造 A 镜像的 Dockerfile 文件中的 ONBUILD 指令就生效了，在构建 B 镜像的过程中，首先会执行 ONBUILD 指令指定的指令，然后才会执行其它指令。

需要注意的是，如果是再利用 B 镜像构造新的镜像时，那个 ONBUILD 指令就无效了也就是说只能再构建子镜像中执行，对孙子镜像构建无效。其实想想是合理的，因为在构建子镜像中已经执行了，如果孙子镜像构建还要执行，相当于重复执行，这就有问题了。利用 ONBUILD 指令，实际上就是相当于创建一个模板镜像，后续可以根据该模板镜像创建特定的子镜像，需要在子镜像构建过程中执行的一些通用操作就可以在模板镜像对应的 dockerfile 文件中用 ONBUILD 指令指定。从而减少 dockerfile 文件的重复内容编写。

## 4.4 dockerfile 样例文件：

### 4.4.1 部署 jar 包，运行命令在 run.sh 文件中

```
FROM java-krb:latest
```

```
ENV LANG zh_CN.UTF-8
```

```
WORKDIR /
```

```
RUN mkdir -p /home/merge && mkdir -p /home/logs && touch /home/logs/gc.log
```

```
COPY innsmap-lbdahis-hbase-dayid.jar /
```

```
CMD ["sh", "run.sh"]
```

#### 4.4.2 部署 redis 并启用两个端口

```
FROM redis
```

```
VOLUME /data
```

```
COPY run.sh /
```

```
COPY redis_6379.conf /etc/redis/
```

```
COPY redis_6380.conf /etc/redis/
```

```
RUN mkdir -p /var/log/redis/
```

```
RUN touch /var/log/redis/redis_6379.log
```

```
RUN touch /var/log/redis/redis_6380.log
```

```
CMD ["sh", "run.sh"]
```

#### 4.4.3 部署 mysql

```
FROM mysql
```

```
VOLUME /var/lib/mysql
```

```
ENV MYSQL_ROOT_PASSWORD mysql123
```

```
COPY setup.sh /mysql/setup.sh
```

```
COPY schema.sql /mysql/schema.sql
```

```
COPY privileges.sql /mysql/privileges.sql
```

```
CMD ["sh", "/mysql/setup.sh"]
```

## 5. 镜像与容器命令

### 5.1 镜像相关命令

#### 5.1.1 获取镜像

```
docker pull ubuntu:12.04
```

#### 5.1.2 列出本地镜像

```
docker images
```

#### 5.1.3 把容器保存为镜像

```
docker commit container_id image_name
```

#### 5.1.4 删除本地镜像

```
docker rmi image_name/image_id
```

### 5.1.5 删除为 none 的镜像

```
docker images|grep none|awk '{print $3 }'|xargs docker rmi
```

### 5.1.6 修改镜像名

```
docker tag imageA imageB
```

### 5.1.7 导出镜像为

```
docker save -o imageA imageA.tar
```

### 5.1.8 导入镜像

```
docker load < imageA.tar
```

### 5.1.9 搜索镜像仓库中的镜像

```
docker search centos
```

### 5.1.10 构建镜像(dockerfile 文件在本地目录)

```
docker build -t . imageA
```

### 5.1.11 显示镜像的历史更改记录

```
docker history --no-trunc IMAGE
```

## 5.2 容器相关命令

### 5.2.1 进入容器

```
docker exec -it 容器 ID /bin/bash --privileged(root 权限进入容器)
```

### 5.2.2 复制文件到容器

```
docker cp test.txt 容器 ID:容器路径
```

### 5.2.3 从容器复制文件到本地

```
docker cp 容器 ID:容器文件 本地路径
```

### 5.2.4 停止已退出的 docker

```
docker ps -a | grep "Exited" | awk '{print $1 }'|xargs docker stop
```

### 5.2.5 删除已退出的容器

```
docker docker ps -a | grep "Exited" | awk '{print $1 }'|xargs docker rm
```

### 5.2.6 查看日志

```
docker logs -f -t --tail=100 container
```

### 5.2.7 docker events

Shell 中执行此命令后，可实时监控所有 container 的状态

### 5.2.8 docker 监控命令

```
docker stats container
```

### 5.2.9 查看镜像或容器的信息

```
docker inspect imageA/containerA
```

### 5.2.10 查看容器的端口映射

```
docker port container
```

### 5.2.11 显示所有运行和停止的 docker

```
docker ps -a
```

### 5.2.12 通过 filter 查找退出状态值为 0 的所有 docker

```
docker ps -a --filter 'exited=0'
```

### 5.2.13 重启停止的 docker

```
docker restart CONTAINER
```

## 5.3 docker run 命令

格式：

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

**d:** 后台运行容器，并返回容器 ID；

**-i:** 以交互模式运行容器，通常与 **-t** 同时使用；

**-p:** 端口映射，格式为：主机(宿主)端口:容器端口

**-t:** 为容器重新分配一个伪输入终端，通常与 **-i** 同时使用；

**--name="nginx-lb":** 为容器指定一个名称；

**-h "mars":** 指定容器的 hostname；

**-e username="ritchie":** 设置环境变量；

**-m:** 设置容器使用内存最大值；

**--net="bridge":** 指定容器的网络连接类型，支持 bridge/host/none/container: 四种类型；

**--privileged:** 特权模式，赋予容器所有的权限

### 5.3.1 docker 镜像 nginx:latest 后台启动 nginx

```
docker run --name mynginx -d nginx:latest
```

5.3.2 使用镜像 `nginx:latest` 以后台模式启动一个容器, 并将容器的 80 端口映射到主机随机端口。

```
docker run -P -d nginx:latest
```

5.3.3 使用镜像 `nginx:latest`, 以后台模式启动一个容器, 将容器的 80 端口映射到主机的 8000 端口, 主机的目录 `/data` 映射到容器的 `/data`。

```
docker run -p 8000:80 -v /data:/data -d nginx:latest
```

5.3.4 绑定容器的 8080 端口, 并将其映射到本地主机 `127.0.0.1` 的 80 端口上。

```
docker run -p 127.0.0.1:80:8080/tcp ubuntu bash
```

5.3.5 使用镜像 `nginx:latest` 以交互模式启动一个容器, 在容器内执行 `/bin/bash` 命令。

```
runoob@runoob:~$ docker run -it nginx:latest /bin/bash
root@b8573233d675:/#
```

## 5.4 docker 系统管理命令

5.4.1 查看 docker 磁盘使用状况

```
docker system df
```

5.4.2 清理磁盘, 删除关闭的容器、无用的数据卷和网络, 以及 dangling 镜像(即无 tag 的镜像)。

```
docker system prune
```

5.4.3 清理得更加彻底, 可以将没有容器使用 Docker 镜像都删掉。注意, 这两个命令会把你暂时关闭的容器, 以及暂时没有用到的 Docker 镜像都删掉了…所以使用之前一定要想清楚呐。

```
docker system prune -a
```

## 6. compose (以 V3 为例)

Compose 是一个 yaml 文件，定义服务、数据卷和网络，默认的路径是 ./docker-compose.yml (扩展名为 .yaml 或 .yml 都可以)

服务定义包含应用于为该服务启动的每个容器的配置，就像将命令行参数传递给 docker container create 一样。同样，网络 and 卷定义类似于 docker network create 和 docker volume create。

与 docker 容器创建一样，Dockerfile 中指定的选项（例如 CMD，EXPOSE，VOLUME，ENV）在默认情况下是生效的，不需要在 docker-compose.yml 中再次指定它们。

### context:

说明:

包含 Dockerfile 的目录的路径，或者是 git 存储库的 url。当提供的值是相对路径时，它被解释为相对于 Compose 文件的位置。此目录也是发送到 Docker 守护程序的构建上下文。

示例:

build:

```
context: ./dir
```

### args:

ARG 只是在 build 构建过程中使用，构建完成后，变量将会消失和 ENV 有着明显的区别而在 compose 的 yaml 文件中 args 的定义:

首先在 dockerfile 中定义参数

ARG buildno

ARG gitcommithash

然后在 compose 文件中添加定义好的值

build:

```
context: .
```

```
args:
```

```
buildno: 1
```

```
gitcommithash: cdc3b19
```

build:

```
context: .
```

```
args:
```

- buildno=1
- gitcommithash=cdc3b19

## cache\_from

使用 cache\_from 指定构建镜像的缓存

build:

```
context: .  
  
cache_from:    # 指定构建镜像的缓存  
- alpine:latest  
- corp/web_app:3.14
```

## labels

向容器添加元数据，和 Dockerfile 的 LABEL 指令一个意思，格式如下：

build:

```
context: .  
  
labels:  
  com.example.description: "Accounting webapp"  
  com.example.department: "Finance"  
  com.example.label-with-empty-value: ""
```

build:

```
context: .  
  
labels:  
- "com.example.description=Accounting webapp"  
- "com.example.department=Finance"  
- "com.example.label-with-empty-value"
```

## command

格式：

command: bundle exec thin -p 3000

command: ["bundle", "exec", "thin", "-p", "3000"]

说明：

会覆盖 dockerfile 中的 command 命令



## config

授权每个服务对配置的访问权限。

`my_config` 是在顶层配置中自定义的配置，配置文件是定义在当前目录下的 `my_config.txt`。

`my_other_config` 是来源于外部的配置文件，可以是其它 `stack` 创建的，也可以是 `docker config create`：定义的配置文件不能为空

```
version: "3.3"
```

```
services:
```

```
  redis:
```

```
    image: redis:latest
```

```
    deploy:
```

```
      replicas: 1
```

```
    configs:
```

```
      - my_config
```

```
      - my_other_config
```

```
configs:
```

```
  my_config:
```

```
    file: ./my_config.txt
```

```
  my_other_config:
```

```
    external: true
```

## deploy

指定与部署或运行服务相关的配置，

```
version: '3'
```

```
services:
```

```
  redis:
```

```
    image: redis:alpine
```

```
    deploy:
```

```
      replicas: 6
```

```
      update_config:
```

```
        parallelism: 2
```

```
        delay: 10s
```

```
      restart_policy:
```

```
condition: on-failure
```

## labels

指定服务的标签。 这些标签仅在服务上设置，而不是在服务的任何容器上设置。

```
version: "3"
```

```
services:
```

```
  web:
```

```
    image: web
```

```
    deploy:
```

```
      labels:
```

```
        com.example.description: "This label will appear on the web service"
```

如果要在容器上设置标签，请使用 `deploy` 之外的标签键：

```
version: "3"
```

```
services:
```

```
  web:
```

```
    image: web
```

```
    labels:
```

```
      com.example.description: "This label will appear on all containers for the  
web service"
```

## mode

可为 `global`（每个 `swarm` 节点只有一个容器）或者 `replicated`（可有多个容器，默认）

```
version: '3'
```

```
services:
```

```
  worker:
```

```
    image: dockersamples/examplevotingapp_worker
```

```
    deploy:
```

```
      mode: global
```

## placement

指定约束和容器所在的节点位置，

```
version: '3.3'
```

```
services:
```

```
  db:
```

```
image: postgres
deploy:
  placement:
    constraints:
      - node.role == manager
      - engine.labels.operatingsystem == ubuntu 14.04
    preferences:
      - spread: node.labels.zone
```

## resources

限制 **docker** 的内存与 **cpu** 资源。

在以下示例中，**redis** 服务被限制为使用不超过 **50M** 的内存和 **0.50**（**50%**）的可用处理时间（**CPU**），并且预留 **20M** 的内存和 **0.25** 个 **CPU** 时间。

```
version: '3'
```

```
services:
```

```
  redis:
```

```
    image: redis:alpinex
```

```
    deploy:
```

```
      resources:
```

```
        limits:
```

```
          cpus: '0.50'
```

```
          memory: 50M
```

```
        reservations:
```

```
          cpus: '0.25'
```

```
          memory: 20M
```

## restart\_policy

说明：

设置如何重启容器，毕竟有时候容器会意外退出。

**condition**：设置重启策略的条件，可选值有 **none**、**on-failure** 和 **any**（默认：**any**）。

**none**：不重启容器；

**on-failure**：在容器非正常退出时（退出状态非 **0**），才会重启容器

**any**：在容器退出时总是重启容器

**delay**：在重新启动尝试之间等待多长时间，指定为持续时间（默认值：**0**）。

**max\_attempts:** 设置最大的重启尝试次数，默认是永不放弃。

**window:** 在决定重新启动是否成功之前要等待多长时间，默认是立刻判断，有些容器启动时间比较长，指定一个“窗口期”非常重要。

示例：

version: "3"

services:

redis:

image: redis:alpine

deploy:

restart\_policy:

condition: on-failure

delay: 5s

max\_attempts: 3

window: 120s

## replicas

指定在任何给定时间应运行的容器数。

version: '3'

services:

worker:

image: dockersamples/examplevotingapp\_worker

networks:

- frontend

- backend

deploy:

mode: replicated

replicas: 6

## dns

两种格式：

dns: 8.8.8.8

dns:

- 8.8.8.8

- 9.9.9.9

## environment

设置环境变量。你可以使用数组或字典两种格式。只给定名称的变量会自动获取它在 **Compose** 主机上的值，可以用来防止泄露不必要的数据。

设置主机名：

```
environment:  
  HOSTNAME: dubbo-service-provider
```

```
environment:  
  RACK_ENV: development  
  SHOW: 'true'  
  MYSQL_ROOT_PASSWORD: mysql123
```

```
environment:  
  - RACK_ENV=development  
  - SHOW=true  
  - MYSQL_ROOT_PASSWORD= mysql123
```

## extra\_hosts

在 container 中的/etc/hosts 文件中添加主机名与 IP 的映射

```
extra_hosts:  
  - "somehost:162.242.195.82"  
  - "otherhost:50.31.209.229"
```

## networks

定义 network 的名称与类型（compose 文件的最底层），IPAM 配置可指定网段分配

```
networks:  
  app_net:  
    driver: bridge  
    enable_ipv6: true  
    ipam:  
      driver: default  
      config:  
        subnet: 172.16.238.0/24
```

```
subnet: 2001:3984:3989::/64
```

## external

如果设置为 **true**，指定来自 **compose** 外部的网络，网络名称为 **host**，启动 **compose** 文件不会创建新的网络（v3.5 以上网络名称需要用 **name** 来指定），本例使用主机网络：

```
version: '2'
```

```
services:
```

```
  proxy:
```

```
    build: ./proxy
```

```
    networks:
```

```
      - "host"
```

```
networks:
```

```
  host:
```

```
    external: true
```

## secrets

配置 **container** 的密钥，密钥可来自本地文件，也可来源于外部文件

```
secrets:
```

```
  my_first_secret:
```

```
    file: ./secret_data
```

```
  my_second_secret:
```

```
    external: true
```

## 6.2 compose 参考文件



docker-compose.yml



docker-compose\_hz.yml

说明：

以上两个文件属于同一个 stack，启动命令为(stack 名字自定义为 innsmap)：

```
docker stack deploy -c docker-compose.yml innsmap
```

```
docker stack deploy -c docker-compose_hz.yml innsmap
```

## 7. docker compose 命令

### 7.1 部署 docker 集群

```
docker stack deploy -c docker-compose.yml inns_docker
```

### 7.2 列出所有节点

```
docker node ls
```

### 7.3 列出所有 stack

```
docker stack ls
```

### 7.4 列出 stack 中所有任务

```
docker stack ps
```

### 7.5 删除 stack

```
docker stack rm
```

### 7.6 列出 stack 中所有服务

```
docker stack services stack-name
```

### 7.7 查看网卡信息

```
docker network ls
```

### 7.8 更新服务所在的节点

```
docker service update --detach=false --constraint-add 'node.hostname==dcons'  
inns_docker_mysql
```

### 7.9 列出所有的服务

```
docker service ls
```

## 8. 问题汇总

### 8.1 如何批量清理临时镜像文件

```
docker rmi $(docker images | grep "none" | awk '{print $3}')
```

### 8.2 运行 docker 容器，出现如下错误

Cannot connect to the Docker daemon. Is the docker daemon running on this host?

#### 8.2.1 原因

从 0.5.2 开始 docker 的守护进程总是以 root 用户来运行。docker 守护进程绑定的是 Unix 的 socket 而不是一个 TCP 端口。Unix 的 socket 默认属于 root 用户，所以，使用 docker 时必须加上 sudo。

#### 8.2.2 解决方法

1.1 切换至 root 用户去运行 docker 命令：

1.2 把当前用户加到 docker 用户组中：

添加 docker 用户组

```
sudo groupadd docker
```

把自己加到 docker 用户组中

```
sudo gpasswd -a myusername docker
```

### 8.3 docker 无法删除本地镜像

执行命令：docker rmi c61c4aaaf461

报错信息：

Error response from daemon: conflict: unable to delete 5ac53667bacd (must be forced) - image is being used by stopped container cc9decae05cf

# 或者

Error response from daemon: conflict: unable to delete c61c4aaaf461 (cannot be forced) - image has dependent child images

上面的错误分别是 被删除的镜像还有实例（容器）在使用（先删除容器）；删除镜像被其他镜像依赖了（先删除其他镜像）

#### 8.3.1 解决办法

根据 imageID 删除正在使用的容器，或者删除依赖的镜像。



## 8.4 容器中的时区与语言的问题

在 docker 容器中默认时间是 UTC 时间，北京是+8 时区。默认语言为 POSIX，则需在编译 docker image 的时候需要指定语言和时区的环境变量：

ENV LC\_ALL en\_US.UTF-8, ENV TZ=Asia/Shanghai

## 8.5、docker 启动报错 error initializing graphdriver

### 8.5.1 启动报错提示如下：

```
[root@docker ~]# systemctl start docker
Job for docker.service failed because the control process exited with error code. See
"systemctl status docker.service" and "journalctl -xe" for more details.
```

### 8.5.2 查看详细报错信息如下：

```
[root@docker ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: failed (Result: start-limit) since 日 2018-04-22 20:52:39 CST; 5s ago
     Docs: https://docs.docker.com
    Process: 4810 ExecStart=/usr/bin/dockerd (code=exited, status=1/FAILURE)
   Main PID: 4810 (code=exited, status=1/FAILURE)
 4月 22 20:52:39 docker.cgy.com systemd[1]: Failed to start Docker Application Container Engine.
4月 22 20:52:39 docker.cgy.com systemd[1]: Unit docker.service entered failed state.
4月 22 20:52:39 docker.cgy.com systemd[1]: docker.service failed.
4月 22 20:52:39 docker.cgy.com systemd[1]: docker.service holdoff time over, scheduling restart.
4月 22 20:52:39 docker.cgy.com systemd[1]: start request repeated too quickly for docker.service.
4月 22 20:52:39 docker.cgy.com systemd[1]: Failed to start Docker Application Container Engine.
4月 22 20:52:39 docker.cgy.com systemd[1]: Unit docker.service entered failed state.
4月 22 20:52:39 docker.cgy.com systemd[1]: docker.service failed.
type=coredump msg=pcp:
INFO[0000] serving...
address="/var/run/docker/containerd/docker-containerd-debug.sock"
module="containerd/debug"
INFO[0000] serving...
address="/var/run/docker/containerd/docker-containerd.sock" module="containerd/grpc"
INFO[0000] containerd successfully booted in 0.002763s module=containerd
```

根据最后的报错 **Error starting daemon** 得到解决办法

#### 8.5.4 具体解决办法

**vim /etc/sysconfig/docker**

加入如下内容：

**OPTIONS="--selinux-enabled --log-driver=journald --signature-verification=false"**

**vim /etc/docker/daemon.json**

加入如下内容：

```
{
  "registry-mirrors": ["http://4a1df5ef.m.daocloud.io"], # 是用来 pull 容器加速用的，
  跟此次问题无关。
  "storage-driver": "devicemapper" # 解决此次问题
}
```

然后重启 **docker**，顺利解决

## 8.6 iptables failed

### 8.6.1 报错提示如下：

```
[root@controller ~]# docker run -it -P docker.io/nginx
/usr/bin/docker-current: Error response from daemon: driver failed programming external
connectivity on endpoint gloomy_kirch
(10289e7a87e65771da90cda531951b7339bee9cb5953474460451cd48013aff0): iptables
failed: iptables --wait -t nat -A DOCKER -p tcp -d 0/0 --dport 32810 -j DNAT --to-
destination 172.17.0.2:80 ! -i docker0: iptables: No chain/target/match by that name.
```

### 8.6.2 报错原因:Firewalld

CentOS-7 中介绍了 **firewalld**，**firewall** 的底层是使用 **iptables** 进行数据过滤，建立在 **iptables** 之上，这可能会与 **Docker** 产生冲突。

当 **firewalld** 启动或者重启的时候，将会从 **iptables** 中移除 **DOCKER** 的规则，从而影响了 **Docker** 的正常工作。

当你使用的是 `Systemd` 的时候，`firewalld` 会在 `Docker` 之前启动，但是如果你在 `Docker` 启动之后再启动 或者重启 `firewalld`，你就需要重启 `Docker` 进程了。

这是由于在运行这次容器之前，成功启动过一次，在上次访问时，因为防火墙的问题导致不能正常访问 `Nginx`，所以将 `iptables` 的 `filter` 表清空了，并且重启过 `iptables`，然后再次运行时，就报了以上错误。

### 8.6.3 解决办法

重启防火墙

## 8.7 Unable to take ownership of thin-pool

### 8.7.1 报错信息

```
Apr 27 13:51:59 master systemd: Started Docker Storage Setup.
Apr 27 13:51:59 master systemd: Starting Docker Application Container Engine...
Apr 27 13:51:59 master dockerd-current: time="2018-04-27T13:51:59.088441356+08:00" level=warning msg="could not change group /var/run/docker.sock to docker: group docker not found"
Apr 27 13:51:59 master dockerd-current: time="2018-04-27T13:51:59.091166189+08:00" level=info msg="libcontainerd: new containerd process, pid: 20930"
Apr 27 13:52:00 master dockerd-current: Error starting daemon: error initializing graphdriver: devmapper: Unable to take ownership of thin-pool (docker--vg-docker--pool) that already has used data blocks
Apr 27 13:52:00 master systemd: docker.service: main process exited, code=exited, status=1/FAILURE
Apr 27 13:52:00 master systemd: Failed to start Docker Application Container Engine.
Apr 27 13:52:00 master systemd: Unit docker.service entered failed state.
Apr 27 13:52:00 master systemd: docker.service failed
```

执行命令：`rm -rf /var/lib/docker/*`

执行命令：`rm -rf /etc/sysconfig/docker-storage`

执行命令：`lvremove /dev/docker-vg/docker-pool`

执行命令：`docker-storage-setup`

重启 `docker` 即可：`systemctl start docker`

## 8.8 docker 启动报错：

**Error starting daemon: SELinux is not supported with the overlay2 graph driver on this kernel**

编辑配置文件 `vi /etc/sysconfig/docker` 将 `--selinux-enabled` 设置为 `false`

## 8.9问题 docker dead but pid file exists

### 8.9.1 解决方案：

```
yum-config-manager --enable public_ol6_latest
yum install device-mapper-event-libs
```

### 8.9.2. 报 No package python-pip available.

原因：这是因为像 **centos** 这类衍生出来的发行版，他们的源有时候内容更新的比较滞后，或者说有时候一些扩展的源根本就没有。所以在使用 **yum** 来 **search python-pip** 的时候，会说没有找到该软件包。因此为了能够安装这些包，需要先安装扩展源 **EPEL**。

解决方案：

```
yum -y install epel-release
yum -y install python-pip
还是不行说的话：
yum install python-setuptools python-setuptools-devel
pip install docker-registry
```

## 8.10 使用/bin/bash 无法进入容器

### 8.10.1 报错信息

```
[root@master gorilla_wifi]# docker run -it m-innsmap-bi /bin/bash
docker: Error response from daemon: oci runtime error: container_linux.go:265: starting
container process caused "exec: \"/bin/bash\": stat /bin/bash: no such file or directory".
[root@master gorilla_wifi]#
```

### 8.10.2 报错原因

该容器中没有可执行的/bin/bash 命令

### 8.10.3 解决办法

使用/bin/sh 命令，容器中都自带有此命令。

## 8.11 compose 暴漏端口问题

```
ports :8088:80
```

注意：8088 是宿主机的端口，80 是 docker 的端口

## 8.12 docker 中部署的程序刚起来后，来不及看日志及调试 docker 就挂掉了

解决方法：compose 文件加入以下命令，先保证容器启动，然后进入容器后手动运行程序

command:

```
["tail", "-f", "/dev/null"]
```

### 8.13 stack 集群模式启动后，修改 config 标签定义的配置文件，删除单个容器后，再重启该容器发现配置未生效。

原因：stack 模式启动后，config 定义的配置文件会加载到内存，所以修改后重启某个容器是不会生效的。只能全部重启。

config 配置文件可在不同节点之间同步，但是 config 文件必须有内容，否则报错。

### 8.14 dockerfile 中 cmd 命令指定了容器的启动命令，但是 docker run -it container /bin/sh 启动容器后发现 dockerfile 中指定的命令并未运行

原因：docker run \* /bin/bash 命令会覆盖 dockerfile 中定义的启动命令，/bin/sh 也属于 docker run 的命令

### 8.15 redis 启动时的警告优化

- 1、you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent\_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.

- 2、临时解决方法：echo never > /sys/kernel/mm/transparent\_hugepage/enabled

进入 docker 中，执行此命令后，发现如下报错（提示只读文件系统，无法修改）：

```
root@bb5bee7de1d6:/#
```

解决方法：以特权模式进入 docker (docker run 加参数：--privileged)，然后即可修改

### 8.16 在 docker 容器中使用 FTP 客户端的问题

我们在 docker 中部署 ftp 服务后发现无法连接到外部的 ftp 服务器，经过多次测试后，最终发现了问题原因。

测试过程：

- 1、在 10.95.5.251 服务器上启动带有 ftp 客户端的 docker，本地办公电脑启用 ftp 服务端（10.95.5.231）。
- 2、在 10.95.5.251 服务器使用 ftp 连接本地电脑及上传下载均没问题。



```
1 10.95.5.251 x 2 10.95.5.251 x 3 10.95.5.251 x 4 10.95.5.13 x +
[root@master ~]# ftp 10.95.5.231
Connected to 10.95.5.231 (10.95.5.231).
220-FileZilla Server 0.9.60 beta
220-written by Tim Kosse (tim.kosse@filezilla-project.org)
220 Please visit https://filezilla-project.org/
Name (10.95.5.231:root): user
331 Password required for user
Password:
230 Logged on
Remote system type is UNIX.
ftp> ls
227 Entering Passive Mode (10,95,5,231,225,156)
150 Opening data channel for directory listing of "/"
-rw-r--r-- 1 ftp ftp      218878713 May 15 09:56 20180515073501.tar.gz
-rw-r--r-- 1 ftp ftp          39 May 15 09:45 build.sh
-rw-r--r-- 1 ftp ftp      2478 May 15 09:45 config.properties
-rw-r--r-- 1 ftp ftp       221 May 15 09:45 Dockerfile
-rw-r--r-- 1 ftp ftp    17632971 May 15 09:46 ip-ftp.jar
-rw-r--r-- 1 ftp ftp       116 May 15 10:10 run.sh
-rw-r--r-- 1 ftp ftp     65632 May 04 14:54 telnet-0.17-64.el7.x86_64.rpm
226 Successfully transferred "/"
ftp>
```

3、进入 docker 后使用 ftp 连接本地电脑报错，报错信息如下

3.1 连接信息（172.17.0.12 是 docker 的 IP）：

```
1 10.95.5.251 x 2 10.95.5.251 x 3 10.95.5.251 x 4 10.95.5.13 x +
[root@master ip-id]# docker inspect --format '{{.NetworkSettings.IPAddress}}' 3ae3364f0fca
172.17.0.12
[root@master ip-id]# docker exec -it 3ae3364f0fca /bin/sh
# ftp 10.95.5.231
Connected to 10.95.5.231.
220-FileZilla Server 0.9.60 beta
220-written by Tim Kosse (tim.kosse@filezilla-project.org)
220 Please visit https://filezilla-project.org/
Name (10.95.5.231:root): user
331 Password required for user
Password:
230 Logged on
Remote system type is UNIX.
ftp> ls
421 Rejected command, requested IP address does not match control connection IP.
ftp>
```

3.2 服务端的报错日志：

421 Rejected command, requested IP address does not match control connection IP.

意思为：请求连接 IP 与控制连接 IP 不同导致连接被拒绝。（宿主机 ip 与 docker 的 ip 不一致）

控制连接：IP 为 10.95.5.251（宿主机 IP）

这个连接用于传递客户端的命令和服务器端对命令的响应。它使用服务器的 21 端口，生存期是整个 FTP 会话时间。

请求连接：IP 为 172.17.0.12（docker 的 IP）

这个连接用于传输文件和其它数据，例如：目录列表等。这种连接在需要数据传输时建立，而一旦数据传输完毕就关闭，每次使用的端口也不一定相同。

```

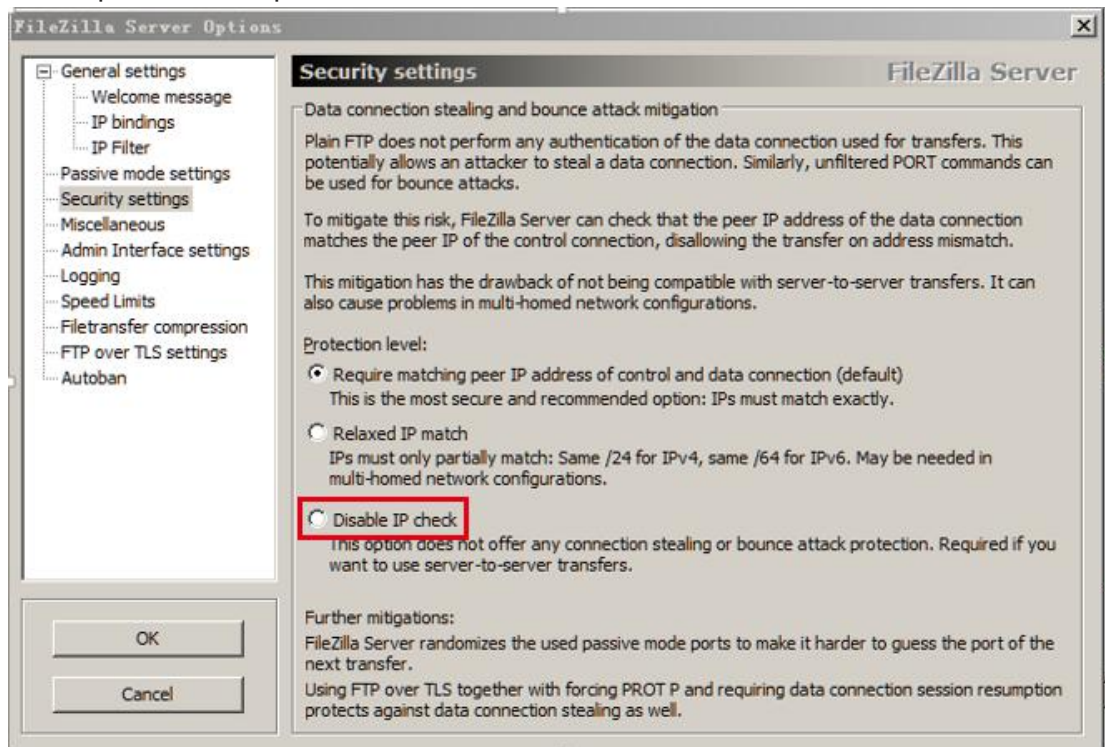
(000210)2018/5/16 14:10:25 - user (10.95.5.251)> 221 Goodbye
(000210)2018/5/16 14:10:25 - user (10.95.5.251)> disconnected.
(000211)2018/5/16 14:12:19 - (not logged in) (10.95.5.251)> Connected on port 21, sending welcome message...
(000211)2018/5/16 14:12:19 - (not logged in) (10.95.5.251)> 220-FileZilla Server 0.9.60 beta
(000211)2018/5/16 14:12:19 - (not logged in) (10.95.5.251)> 220-written by Tim Kosse (tim.kosse@filezilla-project.org)
(000211)2018/5/16 14:12:19 - (not logged in) (10.95.5.251)> 220 Please visit https://filezilla-project.org/
(000211)2018/5/16 14:12:21 - (not logged in) (10.95.5.251)> USER user
(000211)2018/5/16 14:12:21 - (not logged in) (10.95.5.251)> 331 Password required for user
(000211)2018/5/16 14:12:22 - (not logged in) (10.95.5.251)> PASS ****
(000211)2018/5/16 14:12:22 - user (10.95.5.251)> 230 Logged on
(000211)2018/5/16 14:12:22 - user (10.95.5.251)> SYST
(000211)2018/5/16 14:12:22 - user (10.95.5.251)> 215 UNIX emulated by FileZilla
(000211)2018/5/16 14:12:23 - user (10.95.5.251)> PORT 172,17,0,12,123,243
(000211)2018/5/16 14:12:23 - user (10.95.5.251)> 421 Rejected command, requested IP address does not match control connection IP.
(000211)2018/5/16 14:12:23 - user (10.95.5.251)> disconnected.

```

ID	Account	IP	Transfer	Progress	Speed
----	---------	----	----------	----------	-------

### 3.3 解决方法

#### 3.3.1 ftp 服务端有个 ip 检测的设置，可以关闭此设置（disable IP check）



#### 3.3.2 stack 模式使用 host 网络模式（与宿主机 IP 相同）启动 docker,官方说明 compose v3 中的 host 配置方式(具体配置见本文 external)

## HOST OR NONE

Use the `host`'s networking stack, or no networking. Equivalent to `docker run --net=host` or `docker run --net=none`. Only used if you use `docker stack` commands. If you use the `docker-compose` command, use `network_mode` instead.

The syntax for using built-in networks like `host` and `none` is a little different. Define an external network with the name `host` or `none` (that Docker has already created automatically) and an alias that Compose can use ( `hostnet` or `nonet` in these examples), then grant the service access to that network, using the alias.

```
services:
  web:
    ...
    networks:
      hostnet: {}

networks:
  hostnet:
    external: true
    name: host
```

3.3.3 使用 `docker run --network=host` 运行 `docker` 可解决上述问题。

## 3.4 如果采用主动模式

主动模式工作的原理：客户端随机开放一个端口（1024 以上），发送 `PORT` 命令到 FTP 服务器，告诉服务器客户端采用主动模式并开放端口；FTP 服务器收到 `PORT` 主动模式命令和端口号后，通过服务器的 20 端口和客户端开放的端口连接，发送数据。

即：**ftp** 服务端会连接到 **docker** 随机开放的端口，但是此端口无法预料，从而也无法向宿主机暴露，所以 **ftp** 会连接失败（本地测试也是连接失败）。

# 9. 注意事项

## 9.1 不要在容器中存储数据

容器可能被停止，销毁，或替换。一个运行在容器中的程序版本 **1.0**，应该很容易被 **1.1** 的版本替换且不影响或损失数据。有鉴于此，如果你需要存储数据，请存在卷中，并且注意如果两个容器在同一个卷上写数据会导致崩溃。确保你的应用被设计成在共享数据存储上写入。

## 9.2 不要发布两份应用

一些人将容器视为虚拟机。他们中的大多数倾向于认为他们应该在现有的运行容器里发布自己的应用。在开发阶段这样是对的，此时你需要不断地部署与调试；但对于质量保证与生产中的一个连续部署的管道，你的应用本该成为镜像的一部分。记住：容器应该保持不变。

## 9.3 不要创建超大镜像



一个超大镜像只会难以分发。确保你仅有运行你应用/进程的必需的文件和库。不要安装不必要的包或在创建中运行更新（yum 更新）

## 9.4 不要使用单层镜像

要对分层文件系统有更合理的使用，始终为你的操作系统创建你自己的基础镜像层，另外一层为安全和用户定义，一层为库的安装，一层为配置，最后一层为应用。这将易于重建和管理一个镜像，也易于分发

## 9.5 不要使用运行中的容器创建镜像

换言之，不要使用“`docker commit`”命令来创建镜像。这种创建镜像的方法是不可重现的也不能版本化，应该彻底避免。始终使用 `Dockerfile` 或任何其他的可完全重现的 S2I（源至镜像）方法。

## 9.6 不要只使用“最新”标签

最新标签就像 Maven 用户的“快照”。标签是被鼓励使用的，尤其是当你有一个分层的文件系统。你总不希望当你 2 个月之后创建镜像时，惊讶地发现你的应用无法运行，因为最顶的分层被非向后兼容的新版本替换，或者创建缓存中有一个错误的“最新”版本。在生产中部署容器时应避免使用最新。

## 9.7 不要在单一容器中运行超过一个进程

容器能完美地运行单个进程（http 守护进程，应用服务器，数据库），但是如果你不止有一个进程，管理、获取日志、独立更新都会遇到麻烦。

## 9.8 不要在镜像中存储凭据，要使用环境变量

不要将镜像中的任何用户名密码写死。使用环境变量来从容器外部获取此信息。

## 9.9 使用非 root 用户运行进程

“docker 容器默认以 root 运行。（...）随着 docker 的成熟，更多的安全默认选项变得可用。现如今，请求 root 对于其他人是危险的，可能无法在所有环境中可用。你的镜像应该使用 USER 指令来指令容器的一个非 root 用户来运行。

## 9.10 不要依赖 IP 地址

每个容器都有自己的内部 IP 地址，如果你启动并停止它地址可能会变化。如果你的应用或微服务需要与其他容器通讯，使用任何命名与（或者）环境变量来从一个容器传递合适信息到另一个。